Heaps

Written by Rachel Singh, last updated Nov 26, 2018

About

Heaps are another sort of data structure, often visualized as trees but often *implemented* with an array.

Heaps are **complete binary trees** and are an effective way to create a priority queue.

Topics

1. What is a heap?
2. Array-based implementation
3. Heap operations
4. Efficiency

What is a heap?

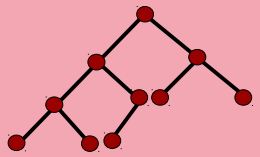
1. What is a heap?

A **(max) heap** is a complete binary tree that either is empty or whose root…

- Contains a value greater than or equal to the value in each of its children, and
- Has heaps as its subtrees.

From Data Abstraction & Problem Solving Walls and Mirrors, 7th ed, by Carrano & Henry, pg 516

Notes



A complete binary tree of height ***h***…

- All nodes at level ***h – 2*** have two children each.
- When a node at level ***h – 1*** has children, all sibling nodes to its left have two children, and
- When a node at level ***h – 1*** has one child, it is a left child.

From Data Abstraction & Problem Solving Walls and Mirrors, 7th ed, by Carrano & Henry, pg 451

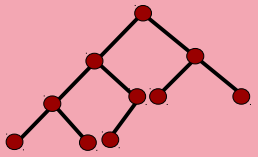
1. What is a heap?

A heap differs from a binary search tree in the following ways:

- You can view a binary search tree as sorted. A heap is ordered in a weaker sense (not strictly sorted).
- Binary search trees can be different shapes. Heaps are always complete binary trees.

From Data Abstraction & Problem Solving Walls and Mirrors, 7th ed, by Carrano & Henry, pg 516

Notes



A complete binary tree of height ***h***…

- All nodes at level ***h – 2*** have two children each.
- When a node at level ***h – 1*** has children, all sibling nodes to its left have two children, and
- When a node at level ***h – 1*** has one child, it is a left child.

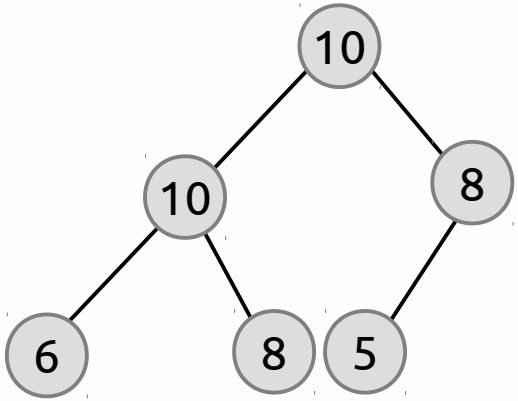
From Data Abstraction & Problem Solving Walls and Mirrors, 7th ed, by Carrano & Henry, pg 451

1. What is a heap?

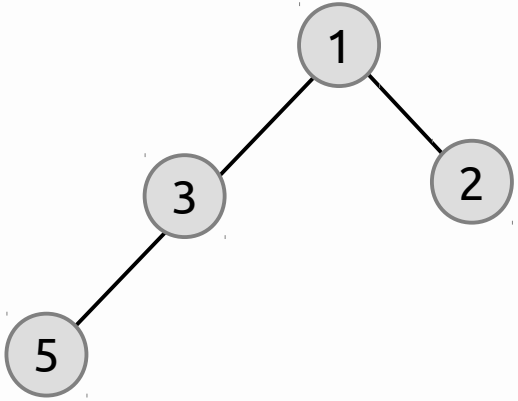
A heap can be:

- a **maxheap** (parent nodes are >= child nodes)
- a **minheap** (parent nodes are <= child nodes)

And this is true for all nodes in the heap.

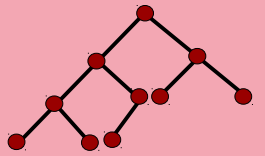


Maxheap



Minheap

Notes



A <u>complete binary tree</u> of height ***h***…

- All nodes at level ***h – 2*** have two children each.
- When a node at level ***h – 1*** has children, all sibling nodes to its left have two children, and
- When a node at level ***h – 1*** has one child, it is a left child.

From Data Abstraction & Problem Solving Walls and Mirrors, 7th ed, by Carrano & Henry, pg 451

Array-based Implementation of a Heap

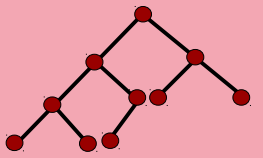
2. Array-based Implementation

Often, heaps are implemented as arrays, even though they're visualized as trees.

In an array-based implementation…

- the **root** is at position 0.
- For every node at index *n*, its left child is at index `[2*n+1]`
- For every node at index *n*, its right child is at index `[2*n+2]`
- For every node at index *n*, its parent is at index `[(n-1)/2]`

Notes



A heap is…

- A complete binary tree
- For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

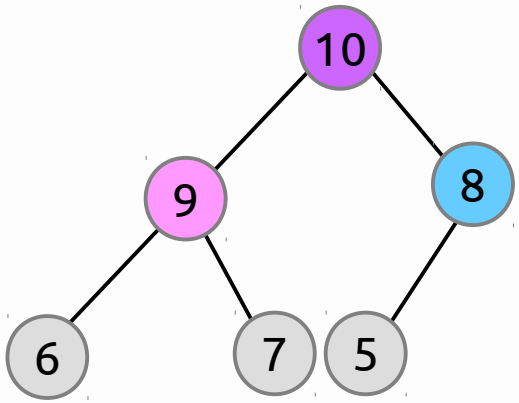
2. Array-based Implementation

For example:

Root (10) is at 0.
Left child is at 2*0+1 = 1
Right child is at 2*0+2 = 2

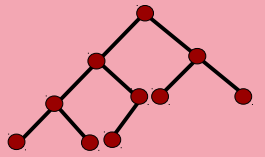
Index	Value
0	10
1	9
2	8
3	
4	
5	

Array



Tree

Notes



A heap is…
- A complete binary tree
- For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

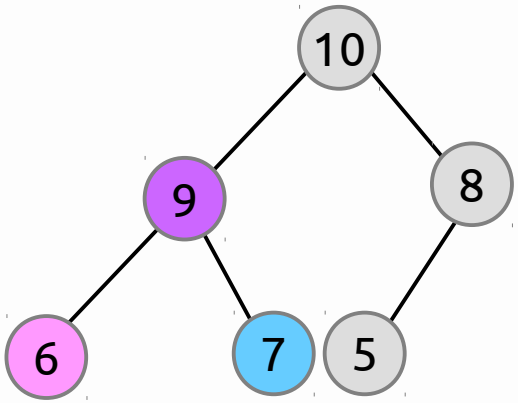
2. Array-based Implementation

For example:

Index	Value
0	10
1	9
2	8
3	6
4	7
5	

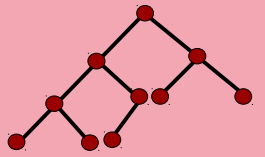
Array

Index 1
Left child is at 2*1+1 = 3
Right child is at 2*1+2 = 4



Tree

Notes



A heap is…
- A complete binary tree
- For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

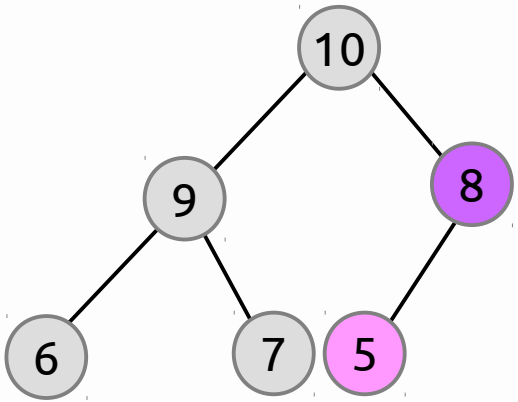
2. Array-based Implementation

For example:

Index	Value
0	10
1	9
2	8
3	6
4	7
5	5

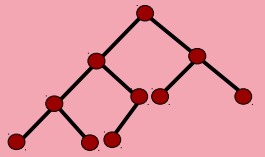
Array

Index 2
Left child is at 2*2+1 = 5



Tree

Notes



A heap is…
- A complete binary tree
- For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

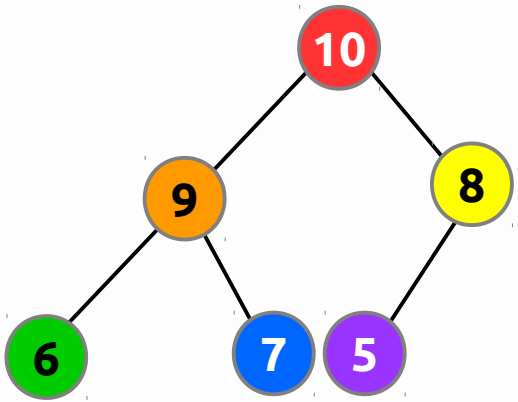
Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

2. Array-based Implementation

For example:

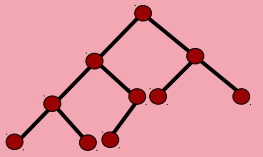
Index	Value
0	10
1	9
2	8
3	6
4	7
5	5

Array



Tree

Notes



A heap is…

- A complete binary tree
- For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:

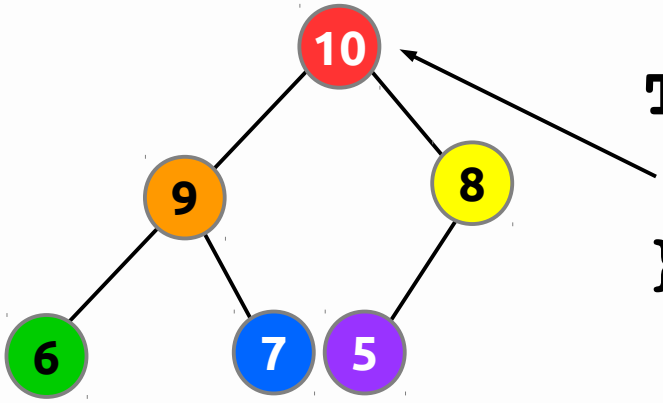
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

Heap Operations

3. Heap Operations

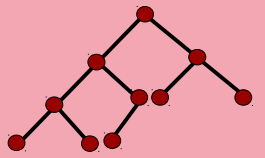
Retrieving an item from the heap:

In a heap, you will just be accessing the max (or min) value – that means, just the root.




```
Top() {
   return heap[0]
}
```

Notes



A heap is…
- A complete binary tree

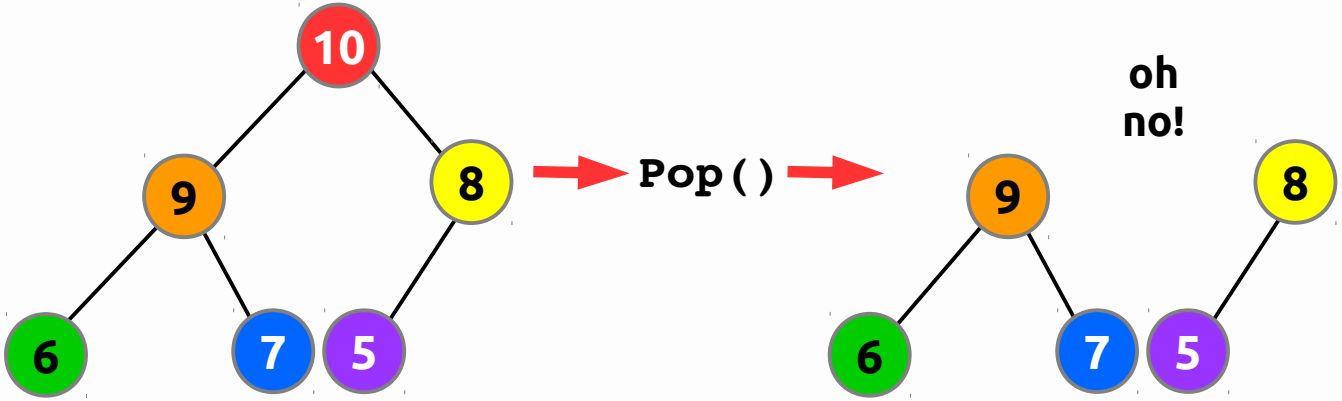
For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

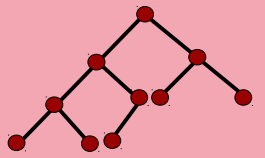
3. Heap Operations

Removing an item from the heap:

When we remove an item from the heap, we will be removing the **root node**. After this operation, we won't have a valid heap anymore: this is called a **semiheap**.



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

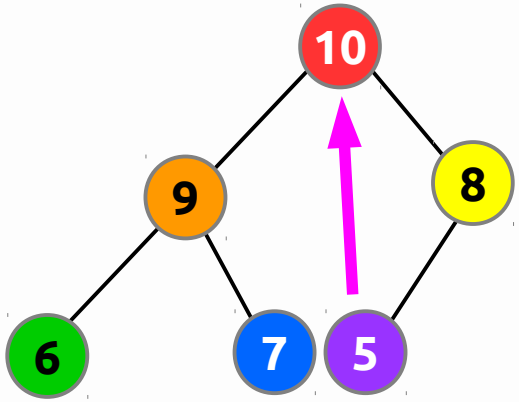
Removing an item from the heap:

Instead of just removing the root node, we're going to look for a replacement.

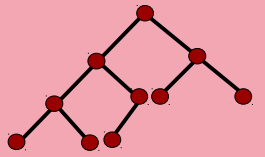
There's only one node we can remove and keep the heap structure – the bottom-most, right-most child.

We will overwrite the root node with this value.

Index	Value
0	10
1	9
2	8
3	6
4	7
5	5



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

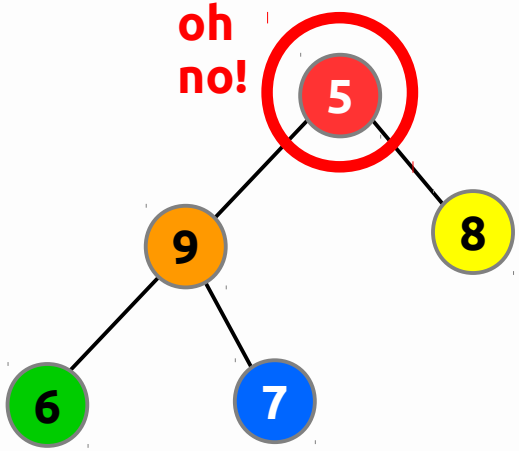
3. Heap Operations

Removing an item from the heap:

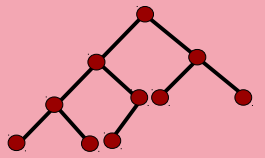
We've replaced the root node (the item we wanted to remove), but now our heap still isn't valid.

This max heap has a 5 as the root! That's *smaller* than its children!

Index	Value
0	5
1	9
2	8
3	6
4	7
5	-



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

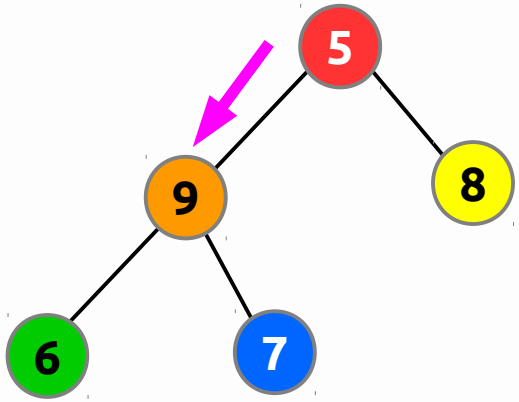
3. Heap Operations

Removing an item from the heap:

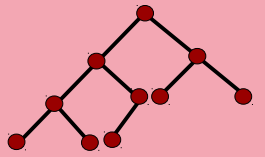
We will need to move the value “5” down the tree (swapping values with other nodes) until it gets to a valid location. This is **percolating down**.

Check both children, and swap with the larger of the children. (or for a minheap, the smallest child.)

Index	Value
0	5
1	9
2	8
3	6
4	7
5	-



Notes



A heap is…
- A complete binary tree

For a maxheap, each node’s value is >= its child nodes’ values; vice-versa for minheap

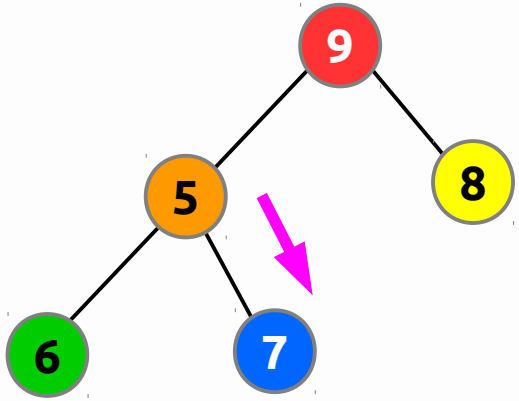
Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

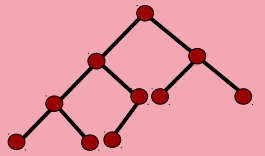
Removing an item from the heap:

Keep doing this until the heap is back in good order.

Index	Value
0	9
1	5
2	8
3	6
4	7
5	-



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

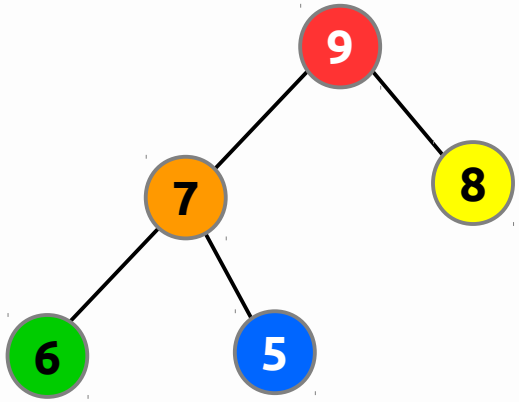
3. Heap Operations

Removing an item from the heap:

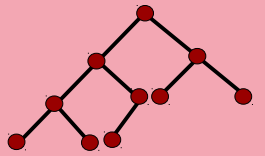
Keep doing this until the heap is back in good order.

OK!

Index	Value
0	9
1	7
2	8
3	6
4	5
5	-



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

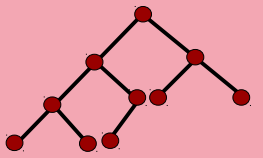
3. Heap Operations

Removing an item from the heap:

```
Pop() {
   // replace root
   heap[0] = heap[size-1]
   size--

   // percolate down:
   while self < children…
      if ( left child > right child )
         swap( self, left child )
      else
         swap( self, right child )
}
```

Notes



A heap is…
- A complete binary tree

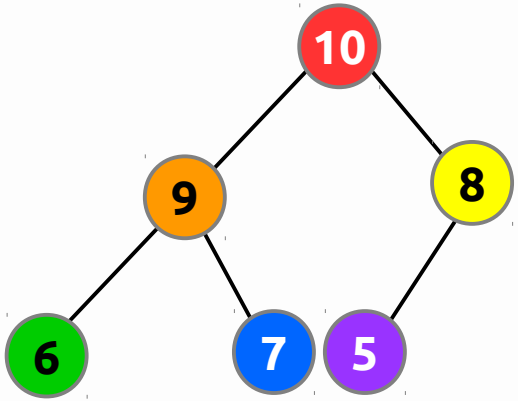
For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

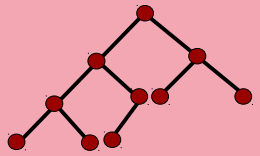
3. Heap Operations

Adding an item to the heap:

When adding an item, remember that we must fill left-to-right and maintain the complete binary tree structure. We must also ensure that the parents always have >= or <= value (max or min heap).



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

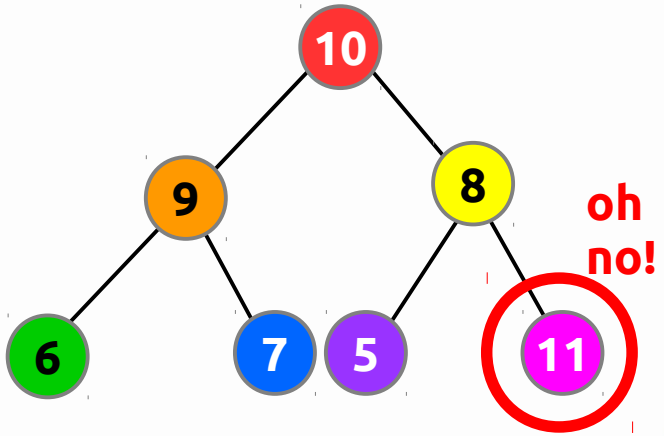
Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

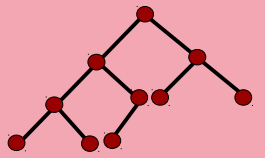
Adding an item to the heap:

We will add our new item to the only valid place: filling the heap from left-to-right. But then our heap will be invalid again.

Index	Value
0	10
1	9
2	8
3	6
4	7
5	5
6	11



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

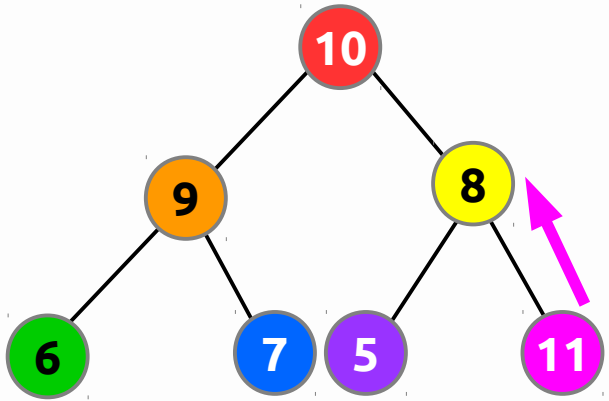
Adding an item to the heap:

Now we need to **bubble up**: Keep moving our new value up the heap as long as

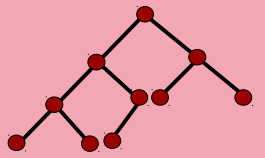
parent < itself

(for a maxheap, vice-versa for a minheap).

Index	Value
0	10
1	9
2	8
3	6
4	7
5	5
6	11



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

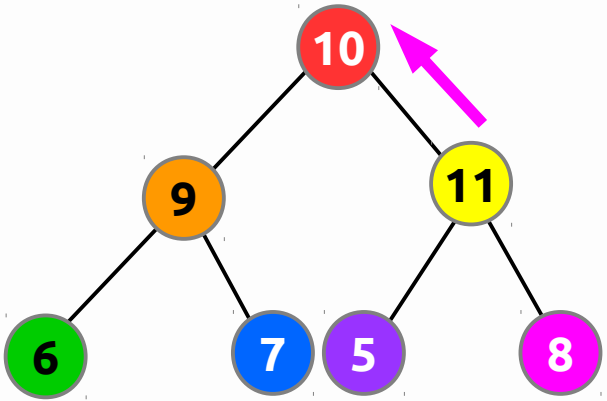
Adding an item to the heap:

Now we need to **bubble up**: Keep moving our new value up the heap as long as

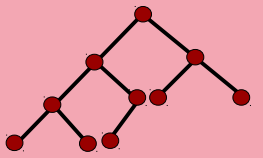
parent < itself

(for a maxheap, vice-versa for a minheap).

Index	Value
0	10
1	9
2	11
3	6
4	7
5	5
6	8



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

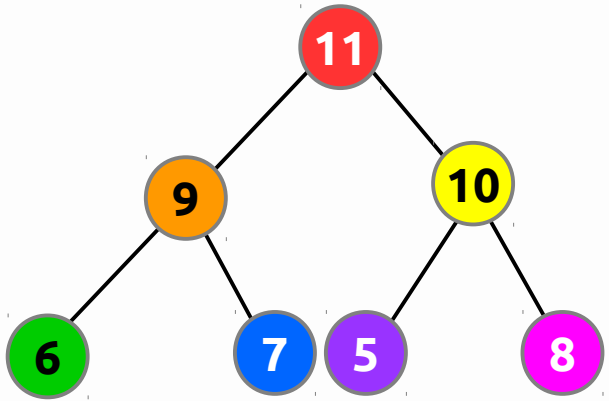
Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

3. Heap Operations

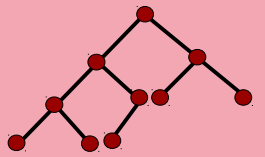
Adding an item to the heap:

Index	Value
0	11
1	9
2	10
3	6
4	7
5	5
6	8

OK!



Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

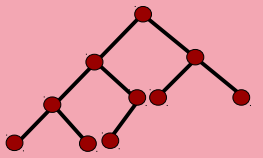
3. Heap Operations

Adding an item to the heap:

```
Add( val ) {
   // Add to end of tree
   heap[size] = val
   size++

   // bubble up:
   while parent < self...
      swap( parent, self )
}
```

Notes



A heap is…
- A complete binary tree

For a maxheap, each node's value is >= its child nodes' values; vice-versa for minheap

Array implementation:
- Root at 0
- Node *n*:
  - Left child at 2n+1
  - Right child at 2n+2
  - Parent at (n-1)/2

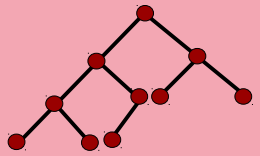
Efficiency of the Heap

4. Efficiency

For a heap…

Function	**Efficiency (Array-based impl)**
Get Top	`O( 1 )`
Pop (then percolate…)	`O( log n )`
Add (then bubble…)	`O( log n )`

Notes



Conclusion

We aren't going to program a heap in this class, though you might do it later on in another course.

The main idea here is to have you understand how they work.